UCRL- 88295
PREPRINT

EXPERIENCE WITH A HIGH ORDER PROGRAMMING LANGUAGE ON THE DEVELOPMENT OF THE NOVA DISTRIBUTED CONTROL SYSTEM*

G. J. SUSKI
F. W. HOLLOWAY
J. M. DUFFY

This paper was prepared for submittal to
5th IFAC Workshop on Distributed
Computer Control Systems
Transvaal, Africa
May 18-20, 1983

May 10, 1983

Lawrence Livermore National Laboratory

EXPERIENCE WITH A HIGH ORDER PROGRAMMING LANGUAGE ON THE DEVELOPMENT OF THE NOVA DISTRIBUTED CONTROL SYSTEM*

G. J. Suski, F. W. Holloway, J. M. Duffy

Lawrence Livermore National Laboratory, University of California,
P. O. Box 5508, L-492, Livermore, California 94550 (USA) PH(415) 422-5917

Abstract. Interest in the impact of high order languages (HOL) on real ~~time and~~ process control applications development has intensified recently, due in part to the Ada language (Ichbiah, 1979) development effort sponsored by the U.S. Department of Defense. High order languages, such as Ada, incorporate state-of-the-art features in the areas of modularity, data abstraction, separate module compilation, strong type checking of data, multi-tasking, distributed processing and exception handling. The intent of such languages is to improve programmer productivity, software maintainability, and the effective management of software development in large real time systems.

This paper explores the impact of an HOL on the development of the distributed computer control system for the Nova laser fusion facility (Simmons, 1982). As the world's most powerful glass laser, Nova will generate 150 trillion watt pulses of infrared light focused onto fusion targets a few millimeters in diameter. It will perform experiments designed to explore the feasibility of fusion as an energy source of the future. Nova will utilize fifty microcomputers and four VAX-11/780's in a distributed process control computer system architecture (Suski, 1982).

Praxis, a high level real time computer language, was designed and implemented for Nova to support the control system development activity (Evans, 1981). Its design is derived from a communications oriented language (COL) which was designed during the period of the early proposals in the Ada language specification process. Praxis is similar to Ada in its design goals, has many features comparable to those found in Ada, and has had operational compilers for over two years.

The purpose of this paper is to assist readers who are beginning to evaluate the importance of HOL's in small to medium scale process control applications. The article begins with a summary of our own application in order to assist readers in applying our experiences to their situations. The history, status and issues relevant to Ada are discussed followed by a brief description of the Praxis development effort. A summary of the major design objectives and features of these HOL's is followed by a description of our experiences with Praxis. Statistics as to the actual utilization of such features are included. The conclusion of this article presents our preliminary expectations and concerns for the use of Ada within process control applications, based upon Praxis experience.

*Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract No. W-7405-ENG-48.

Introduction. Some experts maintain that the dominant programming language of the next decade is destined to be Ada (Carlson, 1981). However the compilers for this language are just now becoming available. The culmination of an eight year effort led by the U.S. Department of Defense, Ada is specifically targeted as a High Order Language (HOL) to meet the requirements of program development and maintenance for embedded real-time control systems. The language offers new and often complex features which are unfamiliar and difficult to evaluate without actual use.

At Lawrence Livermore National Laboratory, a language named Praxis was developed to support the development of software for a fifty computer process control network. This language is similar to ADA in many of its design goals and features. The purpose of this article is to present our experience with Praxis, as a HOL, and our resulting expectations and concerns for Ada in process control applications.

Nova's Distributed Computer Based Control System. The application for which our Praxis-based distributed computer control system was developed is named Nova. It is a 150 terawatt (TW), ten arm laser fusion research facility currently under construction at the Lawrence Livermore National Laboratory (LLNL) (Simmons, 1982)(Fig. 1). As the world's most powerful glass laser system, Nova will provide researchers with an important new tool in the study of inertial confinement fusion (ICF). A principal objective of Nova is to demonstrate the feasibility of generating power from controlled thermonuclear reactions.

An intermediate laser system called Novette (Manes, 1983) utilizing Nova's computer based control system technology, was recently completed at LLNL. Novette utilizes two beams to provide 30 TW of light onto fusion targets. It provides important data on laser operation and target performance in preparation for the Nova experiments, and has proven the Nova control system design.

The Nova system's ten laser beams will be capable of concentrating 100 to 150 kilojoules (kJ) in 3 nanosecond pulses of infrared light onto a fusion target a few millimeters (mm) wide. The system also will generate light at shorter pulse lengths in power bursts up to 150 TW. Nova will also generate green and near ultraviolet light by doubling and tripling the fundamental output wavelength of its amplifiers using passive crystal technology.

All ten arms of Nova must deliver their individual pulses to the target simultaneously (within $\pm$ 5 picoseconds). To achieve this objective, a single 100 microjoule pulse is selected and amplified to approximately 50 joules in a single pass through a nine stage preamplifier. It is then split by partially reflecting mirrors into ten parallel chains of power amplifiers, each consisting of fifteen cascaded laser amplifiers. Each pulse emerges from the output of its 180 meter long chain with a beam diameter of 74 centimeters. The pulses are subsequently reflected by large alignment mirrors, converted to shorter wavelengths, and finally focused onto a fusion target inside a 5 meter diameter aluminum vacuum vessel.

Functional Organization. Nova's control system employs a distributed, computer based architecture which evolved from the successful Shiva laser control system (Suski, 1979). It is organized functionally according to four fundamental subsystems; Power Conditioning, Laser Alignment, Laser Diagnostics, and Target Diagnostics. A fifth, unifying subsystem called Central Controls centralizes, augments, and coordinates the other subsystems' functions (Fig. 2). Criteria of reliability and adaptability are met by the computer based, extendable nature of the system. Flexibility required to optimize individual subsystem architectures is provided by the inclusion of the Central Control subsystem. This subsystem establishes a single point at which compatible interfaces for command, control, and data interactions are established. This architecture supports parallel software development in the five distinct areas, with minimal interaction required between groups.

In this hierarchically structured system, approximately fifty Digital Equipment Corporation (DEC) LSI-11/23 microcomputers provide localized control and data acquisition capabilities in geographically distributed locations throughout the laser fusion facility. Data from these front end processors (FEP's) is collected, analyzed and integrated at the Central Control level with three redundant Digital Equipment Corporation VAX-11/780's minicomputers. Remote command and control capabilities, higher level control functions (e.g., automatic laser alignment), and high volume data storage and manipulation, are implemented at this level. All computers are interconnected using either multi-port memories or a specially developed, high speed (10 Mbits/second) fiber optic network.

Seventy-five man years of effort have been expended in developing the Nova control system including high speed network communications, data base management techniques, real time handlers, operator interfaces, and systems and applications level software.

Why Ada? In the late 1970's, analyses of computer efforts in the United States Department of Defense (DOD) showed that over three billion dollars per year were being expended on software (Fisher, 1978). The majority of this software was used in imbedded real time control systems (e.g., aircraft controls). Issues of maintainability and a need to reduce support costs were steering the DOD towards requiring high order languages in all systems. However, the lack of features and efficiency in these languages led to many exceptions to the use of HOL's. Over fifty percent of all real time imbedded software in DOD systems was being written in assembly language. This resulted in continuous training problems, logistics problems, and upgrade difficulties.

Consequently, the purpose of the DOD in initiating the Ada effort was to develop a language for real time imbedded systems which would be used exclusively on all DOD related projects. The fundamental objectives were:

- Reduce the cost of software throughout its life cycle
- Allow development of truly transportable software - both across applications and hardware

- Allow responsive, timely maintenance of long lived software
- Support very high reliability needs
- Increase readability of all software at the (possible) expense of writability
- Produce high efficiency code, comparable to well coded assembly language.

The History and Current Status of Ada. The Ada language has been under design and development for over eight years. The initial effort was to determine the requirements of a HOL for embedded real-time control systems, and whether existing languages or combinations of these languages could meet these requirements. A series of documents with the code names STRAWMAN, WOODENMAN, and TINMAN were subsequently issued and reviewed by representatives of the academic community, industry, and government. Beginning with STRAWMAN, each successive document became more specific, precise, and complete in stating DOD's HOL design criteria. In 1976, a group of reviewers evaluated existing languages against TINMAN and determined that no existing languages would meet the stated criteria. This committee recommended that a new language be designed, and that it be based on either Algol-68, Pascal, or PL/I.

In 1977, four companies were given contracts to design languages to the IRONMAN specification. Their submittals were given code names and distributed widely for review:

Company	Code
CII-Honeywell	Green
Intermetrics	Red
SRI	Yellow
Softech	Blue

All four companies had based their languages on Pascal. Red and Green were selected as the final contenders. A new and final language requirement document, STEELMAN, clarified and corrected inconsistencies in IRONMAN. Red and Green revised and completed their designs against STEELMAN. After extensive public review, Honeywell's Green was chosen. Contracts for language implementation on Digital Equipment VAX and IBM 370 architectures were awarded. These compilers are expected to be completed in the next twelve months.

With the widespread interest in Ada, additional Ada development efforts around the world have been initiated. This includes a compiler by Intel for the Intel 432, the recently announced ROLM compiler, an interpreter at New York University, and several European, Canadian and Asian efforts.

Ada, however, is to be a tightly enforced standard. Softech, Inc. is completing the Ada validation system which will be used to qualify all compilers using the DOD registered trademark, "Ada". This eliminates the tempting possibility of subsetting the language (to ease implementation) while still referring to such implementations as "Ada".

In addition to the Ada language effort it was recognized that, to achieve Ada's objective, the total programming environment must be considered. It is important to provide the programmer with a standard and sufficient set of tools, independent of the host support system specifics. Therefore, a series of consecutive "environment specifications" including PEBBLEMAN (1978) and subsequently STONEMAN (1980) were developed. At least two major efforts are now underway to implement tools such as editors, libraries, and commonly used run-time support libraries. This work includes the Kennel Ada Programming Support Environment (KAPSE) which presents a virtual host operating system interface to the compiler and other tools. Changes in the actual hardware and operating system can therefore be accommodated within the KAPSE, reducing the need for recoding the support tools for new hosts.

Progress on the environments is not as far advanced as the Ada language itself, yet well designed environmental tools will be required if Ada is to meet its fundamental design objectives.

The current schedule for Ada indicates that DOD compilers and environments should be available in late 1983 and 1984. Mandating the use of Ada in all new DOD embedded real-time applications is planned by 1990.

Ada Concerns. Despite the exhaustive design efforts and considerable commitment to Ada's success, there are still concerns, and opposition to Ada does exist (Ledgard, 1982, Hoare, 1981). The majority of the concerns treat the two categories listed below:

- LANGUAGE COMPLEXITY. The language is too rich and complex, detracting from its usefulness, reliability, and maintainability. Subsets are highly desirable (though specifically prohibited at this time). A result of this complexity is that non-professional programmers may find Ada too intimidating.

- REAL-TIME RESPONSIVENESS. The lack of known storage requirements at compile time and the possible influencing of its high level task synchronization features may impact the safe, reliable use of Ada. This applies specifically to real-time situations such as flight control. Overhead due to run time checking of ranges or other constraints is also a concern.

Until the actual implementations are available, we can only judge Ada by the language definition and not run time performance. Based upon our experience with Praxis, however, we will treat issues of language complexity.

Major Features of Ada. Having summarized the motivation and status of the Ada project, we now briefly review selected important features of the language. Also present in the Praxis language, these features are presented in order according to their impact on the Nova control system development effort:

- Extensive Compile and Run Time Checking - All manipulations involving data, types, and other entities are checked for correctness and legality. This includes the parameters used in procedure calls. Run time checks include range checking of all data values.

- Declarations and User-Defined Data Types - The structured type declarations familiar to Pascal users have been extended to improve capability. Ada (and Praxis) require that the types of all data be declared. Types are strictly enforced and only limited coercion of mixed data types is permitted.

- Separate Compilation - Ada allows procedures, variables, types, and collections thereof to be compiled separately as small units, while maintaining strict compile-time type checking across these units.

- Self-Documenting - The length of identifier names allowed, combined with good data structures and excellent control statements, supports writing of self-documenting software. (It is, of course, still possible to write inscrutable code if such capabilities are not well utilized.)

- Extended Control Structures. - Extensive control structures are provided to clearly indicate operations being performed. Some are redundant. (Praxis does not have the GOTO statement.)

- Packages - Ada provides comprehensive methods of grouping procedures, declarations, and collections of data into separately compilable modules. Ada includes library support for packages and generally improves upon Praxis' capabilities.

- Exception Handling - Exception conditions (I/O interrupts, range errors, underflow) can and must be handled in programmer defined high level code.

- Data Abstraction - Type declarations for data and procedures can be compiled separately from the code which uses or comprises them. This unclutters the user interface and prevents unauthorized changes to data or code.

- Enumerated Data Types - Data types may be defined in which values are limited to a list of programmer defined alphanumeric "names". For example, type color is limited to "values" RED and BLUE.

- Interprocess Communication Constructs - The language provides substantial support for synchronizing processes and interlocking access to shared data.

- Structured Access to Machine Features - Access to machine features is defined and controlled within the language. Frequent use of supporting assembly language programs is unnecessary and discouraged.

Ada includes significant features which were not incorporated onto Praxis due to difficulty of implementation or their limited usefulness. These include:

- o Generic Procedures - Procedures with identical names but different parameter specifications can be utilized and distinguished by the compiler. This increases the value and flexibility of libraries of preprogrammed procedures.
- o I/O - File, text and device oriented I/O is defined as part of the language and implemented in packages.
- o Multi-Tasking - This difficult to implement feature is a major contribution of Ada. Specific constructs for the initiation and control of parallel tasks within the language itself are defined.
- o Standardized Programmer Support Environment - Ada will include standardized tools such as editors, configuration manages, data bases, and user libraries. This feature will also serve to insulate Ada software and tools from changes in operating system software and hardware.

Figures 3 and 4 show examples of procedures written in both Ada and Praxis.

Praxis - Motivation, History and Current Status. We stated at the onset of the Nova Project that substantial savings in time and effort would result if a powerful controls-oriented programming language were available. We had endured several years of dealing with older languages, their restrictions, awkwardness, and inexactness. Extensive debugging sessions often led to discovering a misspelled variable name or a misuse of a variable type. Several different languages and operating system environments had been used since no single product covered the breadth of features needed in this large distributed system. This created typical problems in software maintenance, making support and extension difficult.

Therefore, in January 1979 LLNL issued a contract to Bolt Beranek and Newman (BBN), Inc. to design and to implement a Praxis language compiler for Digital Equipment Corporation PDP-11 computers. The development of Praxis originated from an initial study by BBN, funded by the U.S. Defense Communications Agency (DCA), to determine the requirements of a language for communications programming. With the clarification of the Nova controls hardware architecture and schedule, BBN's work was expanded to include the development of a VAX/VMS native-mode compiler, documentation, additional language design, and a high-level input/output package.

In March 1980 the preliminary PDP-11 compiler successfully passed two critical milestones. The first milestone was that the compiler, which is written in Praxis, complied itself successfully on the PDP-11 systems, proving that the bulk of compiler was correctly implemented.

The second milestone was the implementation of a Nova controls application of the language, a ROM-based LSI-11 processor. A 2000-line assembly-language, stepping motor control program was recoded in Praxis, compiled, and burned into read-only memory (ROM). This demonstrated that the language was indeed powerful enough to replace detailed assembly language sequences and that the compiler correctly implemented the controls-oriented features.

In December 1980 we took final delivery from BBN of the completed Praxis compilers, support software, and documentation. The products were:

- VAX/VMS compiler generating VAX code
- VAX/VMS compiler generating PDP-11 code
- PDP-11/RSX-11M compiler generating PDP-11 code and support software and documentation
- RMS Input/Output package
- Language Reference Manual (Evans, 1981) (300 pages)
- Input/Output Manual
- Compiler Internals document

In addition we completed two in-house documents:

- An Introduction to Praxis
- Programming in Praxis

We have been using the Praxis language for control systems programming since the Summer of 1980 with remarkable success and acceptance. More than 300,000 lines of operational Praxis code have been written.

The Praxis language is specifically within the state of the art of language design. It was particularly designed for control and system implementation needs. It is a comprehensive, strongly typed, block-structured language in the tradition of Pascal, with much of the power of the forthcoming Ada language. It supports the development of systems composed of separately compiled modules, user-defined data types, exception handling, detailed control mechanisms, and encapsulated data and routines. Direct access to machine facilities, efficient bit manipulation, and interlocked critical regions are provided within the language.

Since the control system environment differs in important ways from application to application and machine to machine, Praxis has features to handle these differences. High-level facilities that mask machine dependencies and foster machine independence (portability) usually prevent the use of exactly the programming capability needed for real-time, control applications programming. However, Praxis is a high-level language that has controlled access to machine dependencies.

Complex language features, such as Ada's generic procedures, overloading of operators, and parallel processes, have been intentionally left out. We felt that these concepts were either not understood well enough to be incorporated at this time, or that they need not be part of the language.

Summarizing, Praxis is an extremely powerful, modern programming language that goes beyond Pascal and has been used for over two years. At this time it would be difficult to prove or disprove any cost savings due to the use of Praxis, but a preliminary version of the Nova control system is now in use in the smaller scale version of Nova, Novette, and the system operators are satisfied. The great majority of all of the software for Novette was written in Praxis and the writers are satisfied. Furthermore we have found no application where Praxis was not sufficient causing some other language had to be used.

Experiences with the Praxis HOL. The following information on the actual use of Praxis is based upon personal experience, formal interviews with Nova project programmers, statistical analysis of the 300,000 lines of Praxis source code, and informal communication between all of the Praxis users at LLNL. In presenting the use of these features, we contrast the frequently used and popular ones with those that are impractical, difficult to understand or seldom used. Areas where HOL's like Praxis (and Ada) are difficult to use or whose impact is not yet known, and where additional needs remain are discussed.

Frequently Used Operations in Nova. Predictably, the integer and bit data declarations, and the more traditional flow control operations, were the most frequently utilized statements. Note, however, that while neither Praxis nor Ada contain true string operations, the number of ASCII strings encountered within the Nova control system software was surprisingly large. Approximately 19% of all source code lines contained ASCII strings. A significant portion of these arise from debugging messages in the code. However, a significant portion is also found in software supporting man-machine interactions through operator consoles and other peripheral devices. There should not be any more doubts concerning the importance of string handling operations within control systems. Ada's generic capabilities support the implementation of string operations, and we urge that a standard emerge from the work on environments.

Features and Characteristics Most Liked in Praxis. The availability of Praxis' extensive data structuring facilities was welcomed enthusiastically by our control system implementers. In retrospect, we over utilized this feature and now are retreating. The occurrence of complex uses of large central data structures in software products actually detracts from maintainability.

The single most valuable characteristic of the Praxis language has been the relative completeness with which the compiler can check that the code represents the intent of the programmer. Our experience indicates that once a program has successfully compiled, it will run as expected with little or no machine interactive debugging.

The key features of the language that make this possible are compulsory declarations, enumerations, and tightly enforced type checking. Separate compilation of modules is also a practical requirement.

Large identifier names that are meaningful to the application were judged to be very valuable. It is interesting that this single feature, which is simply and easily implemented in any language, ranks near the top of the list in value to the users. However, for those about to buy programmer workstations, be aware that typical statements in these languages are long. Identifiers in Praxis have a 32 character limit, often used in interests of readability. With block indenting and multi-level structure references, even 132 character wide terminals can be limiting. As a further (minor) comment, the standard eight character wide tab spacing, used by several manufacturers, is too wide and awkward to use with this style of language.

Features Found Impractical for Nova. Impractical features are those which cost more to design, implement or utilize than will be returned in benefit. For example, the designers of Ada and Praxis have attempted to create languages that totally replace the need for assembly language. At least with Praxis, it has been proven to our satisfaction that this is impractical. The effort to implement these features and educate users on their methods and restrictions was, in retrospect, of little net benefit. With the exception of one individual, in-line code is seldomly used and has never been found to be essential.

Another example of an impractical effort is generating machine code rather than intermediate macro assembly code. The Praxis compiler was originally written to output assembly language statements which were then assembled by a standard assembler. This provided an extremely valuable debugging tool - the intermediate assembly language code - which was appreciated by the users. The mysteries of just what the compilers did with exotic source statements, and bugs related to the interrelationship of Praxis code with its environment, were often resolved through close scrutiny of the assembly language code produced by the compiler. However, in the interest of improving overall compilation speed, the Praxis compiler implementers eventually removed this feature from the compiler that generated VAX code. Fortunately, it remains in the LSI-11 code generating version, where it is most useful.

Yet another possibly impractical, but enjoyed, feature of Praxis allowed a carriage return to be a statement terminator instead of a semicolon. This contributed to productivity and readability. However, it used some rather complex rules to determine when end-of-line was actually end-of-statement. As a result, it consumed a considerable amount of time and effort in Praxis compiler design and implementation that may have been better spent testing other features.

Aspects of Praxis and Ada Which Are Difficult to Understand. Ada and Praxis are strongly typed languages. Every use of every identifier is checked for consistency with the original specification of its type. This specification is often in another module (package). This introduces a major complexity in organizing the location of type specifications and in specifying the order in which modules may be compiled.

Frequently, adding a reference to an object used by other modules to a module being modified causes established order of compiling modules to fail. Often, large structural changes must be made to the location (which module) of type definitions, leading to massive editing efforts.

It is often difficult to structure software to meet the requirements introduced by separate compilation and still maintain the flexibility required to easily adapt to changing application requirements. One user has suggested automatic generation of an ordered list of all modules required to be compiled prior to compilation of a given module. One aim of the Ada environment efforts is to treat this specific problem in an automatic or semi-automatic manner.

Seldom Used Features of Praxis. During the design and initial implementation efforts, substantial energies were devoted to theoretical proofs and disproofs leading to the specification and implementation of features. A few such features were included for the sake of elegance, but were often difficult to implement. We found subsequently that such features were often left unused.

While not particularly difficult to implement, one such feature is the ability to combine source statements on a single line separated by semicolons. Since Praxis source lines tend to be lengthy, due to long identifier names and indentation conventions, there are few cases where multiple statements on a single line are desirable. In fact, process control programmers often prefer writing a series of vertically arranged "steps". With the exception of one individual, only 0.6 percent of all source lines contained multiple statements.

Features providing assembly level capabilities were often found to be less useful in a higher level language. For example, the exclusive OR operation, included in many languages, was used exactly three times in the entire system. Another case is the clever SWAP operation where the contents of two variables are exchanged. Often used in assembly level programs in prior systems, SWAP has never been used in Nova.

More importantly, we note that redundant methods of advancing indices on loops and expressing control structures are often not used. A single, simple technique is often selected and employed exclusively.

Features Desired but Not Implemented. The Ada language was designed with three overriding concerns; program reliability and maintenance; concern for programming as a human activity; and efficiency. The need for languages that promote reliability and simplify maintenance is well established. Hence, emphasis was placed on program readability over writeability (Morgan, 1983).

Concern for the human programmer was stressed during the design. But what about the software manager? Praxis did little to assist us with what is the remaining most difficult part of software, its management. While we do not feel qualified to state all that is required, some suggestions follow.

In a loosely organized process control system development effort, a set of compiler controls is required to restrict the use of features which conflict with long term objectives or are otherwise determined to be undesirable in a particular application.

Also of use would be enforced entry of certain information, such as author, project name, date and revision history, organization title and copyright notice. Also desirable is statistical information on the use of various features categorized by author and project as the experience of a novice programmer grows. Many of these areas are being treated in Ada environment.

Better debugging capabilities are needed for Praxis and planned for Ada. In Nova, debug messages are commonly inserted at procedure entry and exit points to show the state of parameters. A compiler should generate these and other messages at designated points according to directives. Higher speed compilation and linking with more aids to regenerating large software products are also desired.

Finally, out of our interview process came a request for a simple initialization operation which could be applied to any data structure. Also requested was that all declared variables default to some known condition. The ability to express the type of storage allocation allowed on parameters to a procedure (for example, must be in I/O space of memory) is desirable. Users would like to see FORTRAN formatted I/O incorporated, but this is not surprising given that I/O in Praxis was not part of the language definition..

Statistical Survey of Features Use Within Praxis. Our Nova control system effort provides a reasonable size sample set for examining the actual use of features within a high order control system language. Several hundred thousand lines of code were written by approximately twenty-five people over a three year time period.

A statistical survey of the use of Praxis features within Nova control system software, including the Praxis compiler, was performed. The incidences of every reserved word and symbol within the language, user defined symbols and types, sizes of modules and lines, and compiler directives were measured. Also recorded was each use of general system software packages. This data was correlated with several categories of users. From these correlations, we determined that users often did not use the very features which they had originally requested.

Clear dialects of usage have developed within small teams. This is taken as evidence that even Praxis, which is sparse compared to Ada, has redundant features. Following the lead of Barnes (Barnes, 1980) we correlated the use of features with several 'cultures' that seemed apparent within the group of developers: Compiler-Writers, Professionals, Amateurs and Novices.

Definition of the Programming Cultures. In our statistical surveys we defined the programming cultures as:

Compiler writers - These are the professionals who specifically design and write language compilers.

Professionals - The professional culture is concerned with writing programs of a permanent nature. These programs are usually large. They are written by teams of individuals whose profession is primarily the design and writing of programs. Their programs should be adequately documented and to accommodate maintenance over a several year lifetime. Accordingly, it is essential that the language used be standard, stable and reasonably well known. Important characteristics are the need for separate compilation, readability and compile time error detection. Interactive use is not required. Examples of existing languages used by this culture are FORTRAN, Pascal, COBOL, PL/I, CORAL 66, RTL/2 and CHILL.

Amateurs - The amateur culture is concerned with writing programs of a less permanent nature. These programs are usually small and often written by individuals whose primary profession is in a different field such as accountancy, medicine, or chemical engineering. They use the computer merely as a tool in the furtherance of their main goals. Their programs are often used only a few times and consequently need little maintenance and documentation. Important characteristics of such languages are the need for ease of writing and general 'user friendliness'; interactive use is inevitable. Examples of such languages are BASIC, FORTH, APL, and the command language of many operating systems.

Novices - People in this category are just beginning to learn about languages and the application of computers.

Table 2 lists some overall statistics correlated with these cultures.

Table 2

	Culture				
	Compiler Writer	Professional	Amateur	Novice	Total
Number of individuals	4	8	9	7	28
Lines of actual code	73,062	109,197	137,043	5,026	324,328
Percentage of lines of comment per line of code	26%	24%	35%	25%	-
Assignments	9,173	11,493	20,426	596	41,688
Number of separate modules	882	661	809	58	2,410
Number of exported items	4,123	6,574	4,204	455	15,356
No. of identifier references	122,712	174,479	226,538	6,750	530,479

Within this summary data some interesting trends are evident. On a percentage basis the amateurs document their code with comments substantially more than the compiler writers and professional programmers (35% versus 25%). The professionals wrote substantially fewer assignment statements and had more module interface declarations, procedures, and type statements. Apparently, they made powerful use of aggregate assignments and procedure calls. The professionals tended surprisingly to write larger modules with more complex interfaces. Amateurs more frequently ventured into unusual features. Experience showed that this caused them some difficulties.

Another statistic of note is the total of over 1/2 million references to identifiers (variables, constants, procedures, functions) required to control the Nova system. Over 4,000 user defined types were employed (see "IS" in Table 2). Also note the difference in the use of OPTIONAL (parameters to procedures and functions), INITIALLY (initial conditions), and EXCEPTIONS by the various cultures.

Projections and Cautions Based on the Praxis Experience on Ada. Our experience with Praxis on the Nova control system indicates that the content and features of the Ada language per se are outstanding. We perceive that the present implementation efforts may be getting ahead of the overall language environmental considerations. Such environmental features as improved compiler and linker speed, interactive capabilities, ease of writing, and general 'user friendliness' should be emphasized. In Nova, the single greatest contribution to our programming style and productivity has been Praxis combined with a screen oriented editor. This reinforces the need to concentrate on providing good Ada environments.

Despite our predictable startup problems with the brand new Praxis language, all of the users reported satisfaction with the overall impact of the language on their projects. Moreover, they intend to use the language, or Ada if it is available, on their next project if possible.

Ada, and to a lesser extent Praxis, are languages rich in syntax and features. There are a sufficient number of redundant features in Ada that dialects will develop. The result in large systems can be that portions of the system written by one team or individual will not be easily understood by other teams or individuals. While sharing of such dialects can be a stimulant to learn, we urge the use of standards of programming style. Such standards should be chosen to be sufficient to limit excessive proliferation of dialects in applications which will require long term maintenance.

Conclusion. We feel that Ada, with good environmental standards, will provide an intellectual stimulant to the advance of professionalism in software development for distributed computer control systems. However, users who plan to use Ada should note that the language by itself will not ensure better, more maintainable control systems. Nor will first time use of Ada on small to medium size systems aid in achieving their timely implementation. The impact of factors such as compiler speed, a rich syntax, strong typing, and separate compilation on initial programmer productivity should be anticipated and factored into project plans.

We end with a positive note: In the Praxis and we expect also in Ada, programs which finally compile successfully often execute successfully the first time. This single characteristic may have a significant impact, in the long term, over the way programs are developed and future languages are designed. Programmers may no longer need to spend most of their development time on the actual control systems. The availability of good, centralized host development facilities will be required. With the use of centralized hosts, the natural communication resulting from the closer proximity of the developers should result in better quality computer control systems in distributed applications such as Nova.

Acknowledgment. We wish to acknowledge the assertiveness and assistance of James Greenwood (former LLNL employee) without whose drive Praxis would not have existed. Also, in particular, Robert Morgan and Art Evans (formally of BBN) for excellence in the compilation, design and implementation of Praxis.

And finally, we thank the following individuals in particular, whose coding styles were analyzed statistically, and who provided the formal and informal material for this paper.

G. Aune	C. Humphreys	J. Severyn
D. Benzel	B. Johnson	T. Sherman
R. Burris	J. Krammen	J. Smart
A. DeGroot	D. Kroepfl	K. Snyder
J. Duffy	D. McGuigan	G. Suski
J. Greenwood	D. Myers	P. VanArsdall
J. Hill	R. Reed	J. Wilkerson
F. Holloway	W. Schaefer	

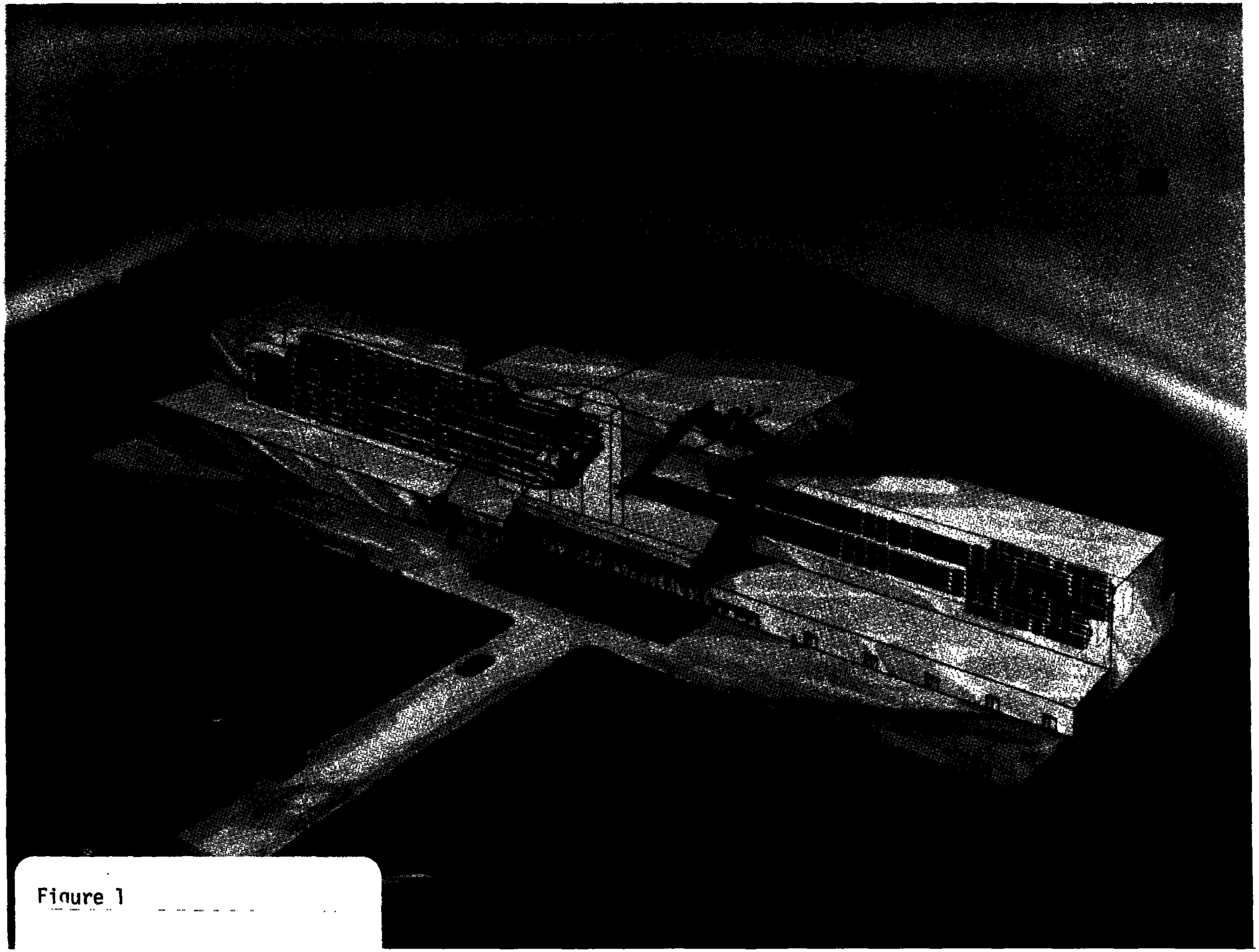

Figure 1

The computer system consists of over 50 computers and associated hardware

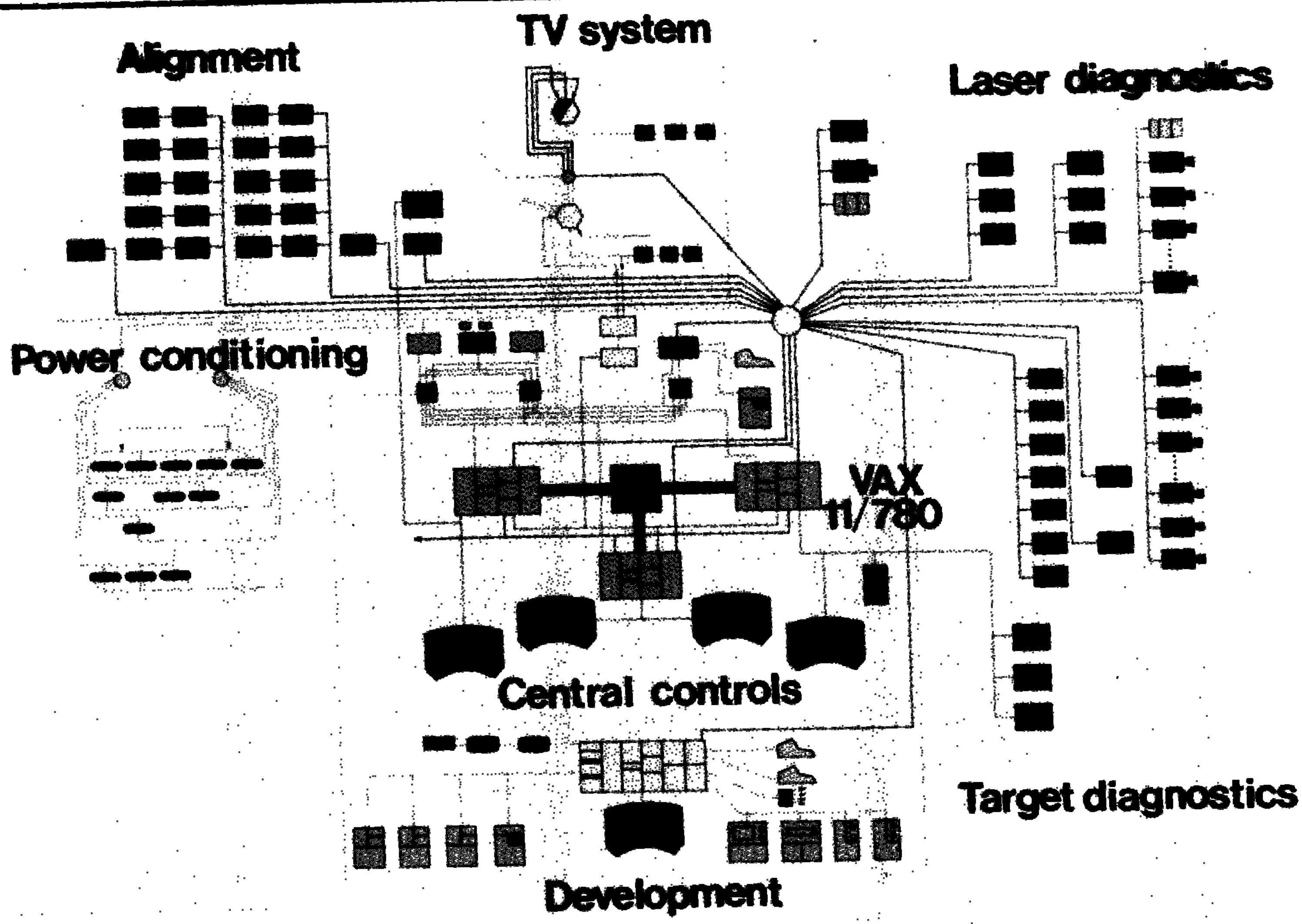

Figure 2

```
//
//      Example of a Complex Number Package in Praxis
//

module COMPLEX_LIB

    use MATH_LIB

    export  COMPLEX, COMPLEX_SUM,
            COMPLEX_PRODUCT, MAGNITUDE

    declare
        COMPLEX is structure
            REAL_PART           : real
            IMAGINARY_PART      : real
        endstructure
    enddeclare

    function COMPLEX_SUM ( X, Y: in ref COMPLEX )
        returns SUM : COMPLEX
        SUM.REAL_PART      := X.REAL_PART + Y.REAL_PART
        SUM.IMAGINARY_PART := X.IMAGINARY_PART + Y.IMAGINARY_PART
    endfunction {COMPLEX_SUM}

    function MAGNITUDE ( X : in val COMPLEX )
        returns R : real initially
        FSQRT ( X.REAL_PART**2 + X.IMAGINARY_PART**2 )
    endfunction {MAGNITUDE}

endmodule {COMPLEX_LIB}

//
// use of above
//

Main Module TEST

    use COMPLEX_LIB

    declare
        X       : COMPLEX
        Y       = COMPLEX ( REAL_PART: 1.0,
                            IMAGINARY_PART: 2.0 )
        Z       : COMPLEX initially Y
        R       : real
    enddeclare

    X := COMPLEX_SUM ( Y, Z )
    R := MAGNITUDE ( X )

endmodule;
```

Figure 3. Comparative Example of Praxis and Ada

```
--
--      Example of a Complex Number Package in Ada
--

package COMPLEX_LIB is

    type COMPLEX is record
        REAL_PART       : real;
        IMAGINARY_PART  : real;
    end record;

    function "+" ( X, Y: COMPLEX ) return COMPLEX ;
    function ABS ( X   : COMPLEX ) return COMPLEX ;

end COMPLEX_LIB;

--
-- And NOW the package body, that is, the implementation
-- of the specification.
--

with MATH_LIB;  -- a package containing FSQRT function

package body COMPLEX_LIB is

begin

    function "+" ( X, Y: COMPLEX ) return COMPLEX is
    -- an overloaded operator
    begin
        return ( X.REAL_PART + Y.REAL_PART,
                 X.IMAGINARY_PART + Y.IMAGINARY_PART);
    end "+";

    function MAGNITUDE ( X   : COMPLEX ) return real is
    begin
        return FSQRT( X.REAL**2 + X.IMAGINARY**2 );
    end MAGNITUDE;


end COMPLEX_LIB;


--
-- use of same in a program fragment
--
    declare
        use COMPLEX_LIB;
        X       : COMPLEX;
        Y       : constant COMPLEX := ( 1.0, 2.0 );
        Z       : COMPLEX := Y;
        R       : REAL;
    begin

        X := Y + Z; -- use complex addition
        R := MAGNITUDE(X);

    end;
```

```
//
//      Message parsing in Praxis
//

module MESSAGE

        export NODE, MSG_TYPE, MSG, SEND_MSG, GET_MSG

    declare

        NODE is [ NORTH, SOUTH, EAST, WEST ]

        MSG_TYPE is [ DATA_MSG, REPLY_MSG, ERROR_MSG, ABORT_MSG ]

        MSG is structure
            TYPE_OF_MSG         : MSG_TYPE initially DATA_MSG
            SOURCE_NODE         : NODE
            DESTINATION_NODE    : NODE
            select TYPE_OF_MSG from
                case DATA_MSG:
                    DATA_VALUE  : real
                case REPLY_MSG:
                    REPLY       : MSG_TYPE
                case ERROR_MSG:
                    STATUS_CODE : integer
                case ABORT_MSG:
                    RE_INIT_FLAG : boolean
            endselect
        endstructure

        HEAP_SIZE       = 100
        MSG_HEAP        : static array [ 1..HEAP_SIZE ]of MSG

    enddeclare

    procedure SEND_MSG( M: MSG, N: NODE)
        //send M to node N
        . . .
    endprocedure

    function GET_MSG ( N: NODE ) returns M: MSG
        // get next MSG from node N
        . . .
    endfunction

endmodule {MESSAGE}
```

Figure 4: Comparative example of Praxis and Ada

```
--
--      Message parsing in Ada
--

package message is

    type NODE is ( NORTH, SOUTH, EAST, WEST );

    type MSG_TYPE is ( DATA_MSG, REPLY_MSG,
                       ERROR_MSG, ABORT_MSG );

    type MSG is record
        TYPE_OF_MSG             : MSG_TYPE := DATA_MSG;
        SOURCE_NODE             : NODE;
        DESTINATION_NODE        : NODE;
        case TYPE_OF_MSG is
            when DATA_MSG =>
                DATA_VALUE      : real;
            when REPLY_MSG =>
                REPLY   : MSG_TYPE;
            when ERROR_MSG =>
                STATUS_CODE     : integer;
            when ABORT_MSG =>
                RE_INIT_FLAG : boolean;
        end case;
    end record;

    procedure SEND_MSG( M: MSG, N: NODE);
            -- send M to node N

    function GET_MSG ( N: NODE ) return MSG;
            -- get next MSG from node N

end MESSAGE;

--
-- And Now the package body
--

package body MESSAGE is

        HEAP_SIZE : constant 100;
        MSG_HEAP : array ( 1..HEAP_SIZE ) of MSG;

begin

    procedure SEND_MSG( M: MSG, N: NODE) is
            -- send M to node N
    begin
            ...

    end SEND_MSG;

    function GET_MSG ( N: NODE ) return MSG is
            -- get next MSG from node N
    begin
            ...

    end GET_MSG;

end MESSAGE;
```

```
//
//      Use of same in a program (Praxis)
//

main module TEST2

    use MESSAGE
    use TEXTIO  // contains rudamentary i/o

    declare
        IN_MSG          : MSG
        ABORT_FLAG      : boolean initially false
        COUNT           : integer initially 0
    enddeclare

    while not ABORT_FLAG do

        IN_MSG := GET_MSG ( SOUTH )

        select IN_MSG.TYPE_OF_MSG from

            case DA1A_MSG:
                OUT_REAL(tty,IN_MSG.DATA_VALUE)

            case ABORT_MSG:
                ABORT_FLAG := IN_MSG.RE_INIT_FLAG

            case ERROR_MSG:
                out_integer (tty,IN_MSG.STATUS_CODE)

        endselect

        // Now send a response

        SEND_MSG ( MSG ( TYPE_OF_MSG: REPLY_MSG,
                         REPLY  : IN_MSG.TYPE_OF_MSG ),
                   SOUTH )

        COUNT *= + 1

    endwhile

endmodule
```

```
--
--  Use of same in a program fragment (Ada)
--

declare

    use MESSAGE;
    use SIMPLEST;       -- contains rudamentary i/o

    IN_MSG      : MSG;
    ABORT_FLAG  : boolean := false;
    COUNT       : integer := 0;

begin

    while not ABORT_FLAG loop;

        IN_MSG := GET_MSG ( SOUTH );

        case IN_MSG.TYPE_OF_MSG is
            when DATA_MSG =>
                put_real(IN_MSG.DATA_VALUE);

            when ABORT_MSG =>
                ABORT_FLAG := IN_MSG.RE_INIT_FLAG ;

            when ERROR_MSG =>
                put_integer (IN_MSG.STATUS_CODE);
        end case;

        -- now send a response

        SEND_MSG (( REPLY_MSG, IN_MSG.TYPE_OF_MSG ),
                    SOUTH );

        COUNT := COUNT + 1;

    end loop;
end;
```

Figure 4 continued

REFERENCES

Barnes, J. G. P., "An Overview of Ada", Software-Practice and Experience, pp. 851-887, 1980.

Carlson, W. E., "Ada: A Promising Beginning", Computer, 1978.

Evans, A., Jr., C. R. Morgan, J. R. Greenwood, M. C. Zarnstorff, G. J. Williams, E. A. Killian, J. H. Walker, and J. M. Duffy, Praxis Language Reference Manual, University of California UCRL-15331 (1981), Contract 3634909.

Evans, A., Jr., A Comparison of Programming Languages: ADA, PRAXIS, PASCAL, C, UCRL 15346, (1981).

Fisher, D. A., "DOD's Common Programming Language Effort", Computer, 1978.

Hoare, C. A. R., "The Emperor's Old Clothes", Communications of the ACM, pp. 75-83, 1981.

Ichbiah, J. D., J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-brueckner, and B. A. Wichmann, Rationale for the Design of the ADA Programming Language, ACM Sigplan Notices, V. 14, No. 6, June 1979, Part B.

Ledgard, H. F., and A. Singer, "Scaling Down Ada (Or Towards A Standard Ada Subset), Communications of the ACM, pp. 121-125, 1982.

Manes, K. R., et al, Novette, A Short Wavelength Laser-Target Interaction System, UCRL 88120, prepared for submission to the Sixth International Workshop on Laser Interaction & Related Plasma Phenomena (1982).

Morgan, C. R., "ADA: Its Motivation, History, Capabilities, Implementation", Intermetrics Inc. Lecture Notes, 1983.

Preliminary ADA Reference Manual, ACM Sigplan Notices, V. 14, No. 6, June 1979, Part A.

Saib, S. H., and R. E. Fritz, "The ADA Programming Language: A Tutorial", IEEE Computer Society Press, IEEE Catalog EHO 202-2 (1983).

Simmons, W. W., et al, Nova Laser Fusion Facility: Design, Engineering, and Assembly Overview, UCRL-8870, prepared for submission to the Journal of Nuclear Technology/Fusion (1982).

Suski, G. J., and F. W. Holloway, "Development of a Control-System Implementation Language (Praxis)", Laser Program Annual Report, Lawrence Livermore Laboratory, UCRL-50021-79, pp. 2-106.

Suski, G. J., and F. W. Holloway, "The Evolution of a Large Laser Control System - from Shiva to Nova", IEEE Circuits and Systems, Vol. 1, No. 3, Sept. 1979 and cover photo.

Suski, G. J., J. M. Duffy, D. G. Gritton, F. W. Holloway, J. E. Krammen, R. G. Ozarski, J. R. Severyn, and P. J. VanArsdall, The Nova Control System - Goals, Architecture, and System Design, University of California UCRL-86827 (1982).